

# Lesson 8 Red-Black Trees

## Wholeness of the Lesson

Red-black trees provide a solution to the problem of unacceptably slow worst case performance of binary search trees. This is accomplished by introducing a new element: nodes of the tree are colored red or black, adhering to the balance condition for red-black trees. The balance condition is maintained during insertions and deletions and doing so introduces only slight overhead.

**Science of Consciousness:** Red-black trees, as an example of BSTs with a balance condition, exhibit the Principle of the Second Element for solving the problem of skewed BSTs.

# The Goal

- ◆ Improve worst-case performance of BSTs by introducing a balance condition that
  - can be maintained after each insertion and deletion
  - guarantees the height of the tree is always  $O(\log n)$ , where  $n$  is the number of nodes in the tree.
- ◆ Achieving this objective implies that all insertions, deletions and searches have a worst-case running time of  $O(\log n)$ .

# Overview of the Lesson

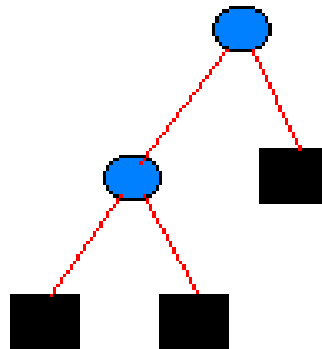
- ◆ Specify the balance condition that defines a red-black tree.
- ◆ Verify that the height of any red-black tree is  $O(\log n)$ .
- ◆ Using a red-black tree as a dictionary.
- ◆ Work through the insertion algorithm for a red-black tree.


# Overview of the Lesson

- ◆ Specify the balance condition that defines a red-black tree.
- ◆ Verify that the height of any red-black tree is  $O(\log n)$ .
- ◆ Using a red-black tree as a dictionary.
- ◆ Work through the insertion algorithm for a red-black tree.

# Tree Terminology And Conventions

1. We always think of our trees as terminating with a NULL reference. This convention will be important when we compute the height of red-black trees. Sometimes the NULL nodes are not shown.



- 
2. External nodes are NULL references; internal nodes are non-NULL nodes; if we refer to a “node” without specifying external or internal, we mean “internal node”.
  3. In the present context, a “leaf node” is an internal node that has only NULL children.
  4. The height of an empty tree is -1.
  5. A path from a node  $x_0$  to a node  $x_k$  is a sequence  $x_0; x_1; \dots ; x_k$  so that  $x_{i+1}$  is a child of  $x_i$  (for  $i = 0; 1; 2; \dots ; k - 1$ ). The length of such a path is the number of predecessors of the last element  $x_k$  (so in this case,  $= k$ ).
  6. The height of a node  $x$  is the length of the longest path from  $x$  to a leaf node. (Note: the height of NULL is -1.)

# Definition of Red-Black Trees

A BST is red-black if it has the following 4 properties:

- A. Every node is colored either red or black
- B. The root is colored black.
- C. If a node is red, its children are black.
- D. For each node  $x$ , every path from  $x$  to a NULL reference has the same number of black nodes. This number is called the black height of  $x$  and is denoted  $bh(x)$ .

Note: The black height of NULL is 0.

# Overview of the Lesson

- ◆ Specify the balance condition that defines a red-black tree.
- ◆ Verify that the height of any red-black tree is  $O(\log n)$ .
- ◆ Using a red-black tree as a dictionary.
- ◆ Work through the insertion algorithm for a red-black tree.



**Theorem.** The height of any red-black tree is  $\leq 2 \log(n + 1)$ , where  $n$  is the number of internal nodes in the tree.

**Lemma 1.** For any red-black tree having height  $h$ ,

$$\text{bh}(r) \geq \frac{h}{2}$$

where  $r$  is the root.

*Intuition.* Though the black height of a red-black tree is typically less than the height of the tree, it is not “too much” smaller — it’s a reasonable approximation.

**Proof.** If the tree is null, this is obvious. Otherwise, follow any path from the root to a deepest leaf. The root is black, and one (or more) of the next two nodes is black, then one (or more) of the two nodes after that are black, and so forth. Thus, at least half the nodes along such a path are black. ■

**Lemma 2.** Suppose  $x$  is any node in  $T$  (internal or external). Let  $n(T_x)$  denote the number of *internal* nodes in the subtree of  $T$  at  $x$ . Then

$$n(T_x) \geq 2^{\text{bh}(x)} - 1.$$

*Intuition.* In a *completely filled* binary tree, we know that  $n = 2^{h+1} - 1$ . The idea is that when a tree is balanced, it must branch a lot and its number of nodes must be exponentially bigger than its height. Since black height is an approximation to height, therefore, the lemma says that “since red-black trees are highly balanced, the number of nodes is always exponentially larger than the height.”

⊕ **Proof.** Proceed by induction on the height  $h(x)$  of  $x$  in  $T$ .

- If  $h(x) = -1$ , then  $x$  must be null. It follows  $\text{bh}(x) = 0 = n(T_x)$  and the result follows.
- If  $h(x) = 0$ , then  $x$  is a leaf node, so  $\text{bh}(x) = 0$  or  $1$ . Again

$$2^{\text{bh}(x)} - 1 \leq 1 = n(T_x)$$

- For the induction step, assume the result holds true for heights  $< h$ , where  $h \geq 1$ . Let  $x_L$  and  $x_R$  be left and right children of  $x$ .

*Claim.*  $h(x_L) < h(x)$  and  $h(x_R) < h(x)$

*Proof of Claim.* We show it for  $x_L$ ; the other case is similar. If  $x_L$  is null, then  $h(x_L) = -1 < 1 \leq h(x)$ . If  $x_L$  is not null, then by definition of height,  $h(x_L) = h(x) - 1 \leq h(x)$ .

*Continuation of proof of theorem.* By the claim, the induction hypothesis can be applied to  $T_{x_L}$  and  $T_{x_R}$ :

$$\begin{aligned} (*) \quad n(T_{x_L}) &\geq 2^{\text{bh}(x_L)} - 1 \\ n(T_{x_R}) &\geq 2^{\text{bh}(x_R)} - 1 \end{aligned}$$

Notice also that if  $x$  is red,  $\text{bh}(x_L) = \text{bh}(x)$  (likewise for  $x_R$ ), whereas if  $x$  is black,  $\text{bh}(x_L) = \text{bh}(x) - 1$  (likewise for  $x_R$ ). Therefore

$$\begin{aligned} (**) \quad \text{bh}(x_L) &\geq \text{bh}(x) - 1 \\ \text{bh}(x_R) &\geq \text{bh}(x) - 1 \end{aligned}$$

Putting (\*) and (\*\*) together, we have

$$\begin{aligned} n(T_x) &= n(T_{x_L}) + n(T_{x_R}) + 1 \\ &\geq 2^{\text{bh}(x_L)} - 1 + 2^{\text{bh}(x_R)} - 1 + 1 \quad (\text{by } (*)) \\ &\geq 2^{\text{bh}(x)-1} - 1 + 2^{\text{bh}(x)-1} - 1 + 1 \quad (\text{by } (**)) \\ &\geq 2 \cdot 2^{\text{bh}(x)-1} - 1 \\ &= 2^{\text{bh}(x)} - 1. \end{aligned}$$

**Proof of Theorem.** Let  $h = h(T)$  denote the height of  $T$ . We wish to show  $h(T) \leq 2 \log(n+1)$ . Suppose this fails to be true so that

$$(1) \quad h(T) > 2 \log(n+1).$$

We arrive at a contradiction: From (1) we get

$$(2) \quad \frac{h(T)}{2} > \log(n+1).$$

Let  $r$  denote the root node. From Lemma 1 we get

$$(3) \quad \text{bh}(r) \geq \frac{h}{2}$$

Applying exponentiation to (2) and (3) in succession:

$$2^{\text{bh}(r)} \geq 2^{\frac{h}{2}} > n+1.$$

It follows that

$$n < 2^{\text{bh}(r)} - 1.$$

This contradicts Lemma 2. ■

# Main Point

Because of their balance condition, red-black trees always have height  $O(\log n)$ , so their primary operations all have running times that are  $O(\log n)$ . Techniques for maintaining the red-black properties after insertions and deletions are techniques that maintain balance in the midst of change.

Science of Consciousness: "Far away indeed from the balanced intellect is action devoid of greatness." (Gita, II.49) The "balanced intellect" is a state of life in perfect balance in which each area of life from most expressed to most subtle and refined is spontaneously given due attention. Such a life is filled with great accomplishment and success. The technique to maintain balance is regular contact with the Self, the field of pure consciousness; having contacted this field, awareness is able to maintain a profound state of balance even in the context of active life.

# Overview of the Lesson

- ◆ Specify the balance condition that defines a red-black tree.
- ◆ Verify that the height of any red-black tree is  $O(\log n)$ .
- ◆ Using a red-black tree as a dictionary.
- ◆ Work through the insertion algorithm for a red-black tree.

# Using a Red-Black Tree As a Dictionary

1. A dictionary is a collection of pairs  $(k, e)$ , which we consider to be key/value pairs. The purpose of a dictionary is to make it possible to look up values by key.
2. The Dictionary ADT supports these operations:
  - `get(k)` - returns the value  $e$  that is paired with  $k$
  - `put(k,e)` - inserts the pair  $(k, e)$  into the dictionary
  - `remove(k)` - removes any pair in the dictionary whose key is  $k$
  - `containsKey(k)` - returns true if  $k$  occurs as a key in the dictionary



# Implementations of Dictionaries

Dictionaries can be implemented in different ways, depending on the purpose.

- a) List. Key/value pairs can be inserted into any kind of list. Insertions can be done by adding to the end of the list in  $O(1)$  time. However, lookups and deletes require  $O(n)$  time.
- b) Hashtable. As we have seen, hashtables have average case running time  $O(1)$  for all of these operations. However, hashtables do not store keys in any particular order.
- c) Red-black Tree. When keys of key/value pairs have a natural ordering (like integers or strings), key/value pairs can be stored in the nodes of a red-black tree and ordered by keys. The operations for this kind of dictionary are then implemented by invoking the corresponding BST operations. (For example, to implement `get(k)`, one performs a search for the node containing the key `k`; when found, the value `e` stored with `k` is returned.) Red-black tree implementations of Dictionary guarantee  $O(\log n)$  performance of the Dictionary operations.

# Ordered Dictionary

An ordered dictionary is a dictionary that maintains its data in sorted order, sorted by keys. When elements of the dictionary are printed, they appear in sorted order (by the ordering of the keys). A red-black tree implementation of a dictionary is an example of an ordered dictionary. The price that is paid for maintaining elements in sorted order -- a characteristic that hashtables do not have -- is that running times of the operations are only  $O(\log n)$  (instead of  $O(1)$ ). Note: Java's implementation of an ordered dictionary is **TreeMap**.

# Overview of the Lesson

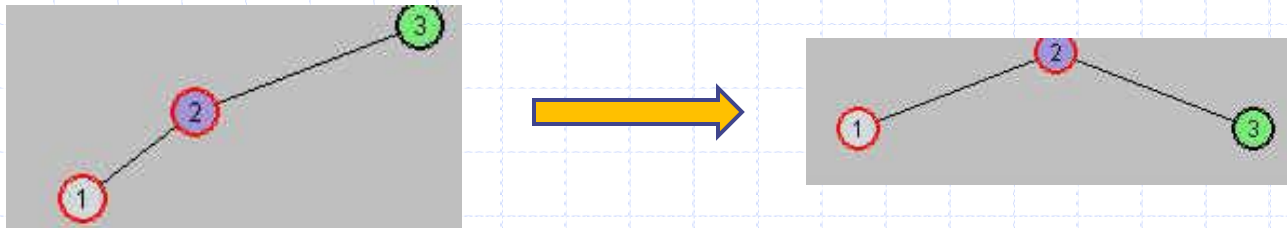
- ◆ Specify the balance condition that defines a red-black tree.
- ◆ Verify that the height of any red-black tree is  $O(\log n)$ .
- ◆ Using a red-black tree as a dictionary.
- ◆ Work through the insertion algorithm for a red-black tree.

# Preparing to Insert into a Red Black Tree: *Rotations*

- ◆ Introducing the primary re-balancing technique:  
*rotations*
  - Used to keep a red-black tree balanced
  - It has the effect of raising the height of some nodes and lowering others, but preserving the BST property

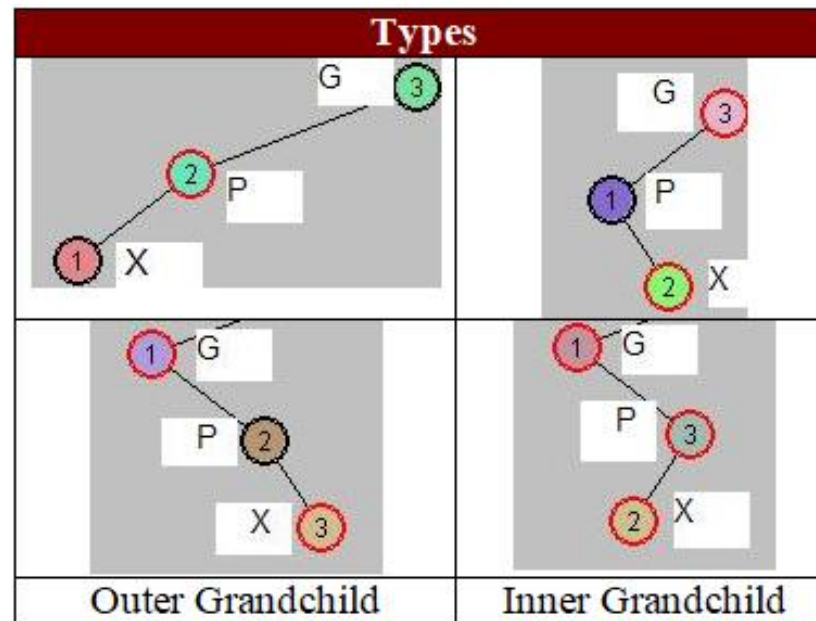
# Examples of Rotations

- a. Perform a right rotation of tree obtained from 3,2,1 insertion. At first, the node 3 is the *top*. After rotation, tree is balanced and still a BST.



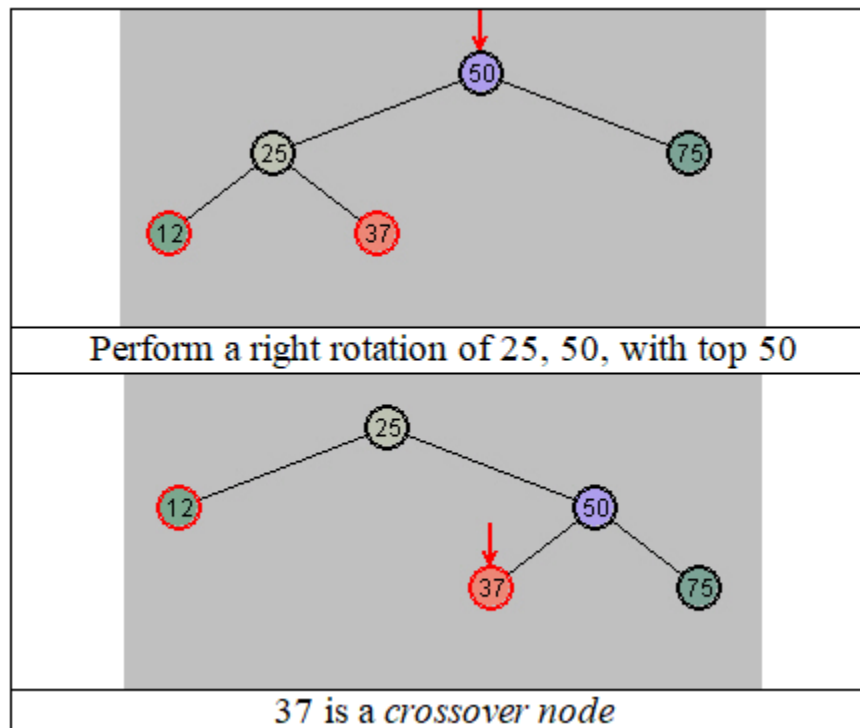
# Examples of Rotations

- b. Rotations involving a grandchild.* Often need to move a node X and the parent P and grandparent G need to be examined. Relative to the grandparent, X can be an *outer grandchild* or an *inner grandchild*.



# Examples of Rotations

- c) In complicated rotations (for instance, in a right rotation where the child being lifted already has a right child), there may be a *crossover node*.



# Top-Down Insertion in a Red-Black Tree

## 1. Basic idea:

- a. To insert a new node, attempt to locate proper location using usual BST algorithm
- b. On the way down the tree, make “preventive” adjustment
- c. Perform the insertion
- d. Correct any red-red violations near the insertion point

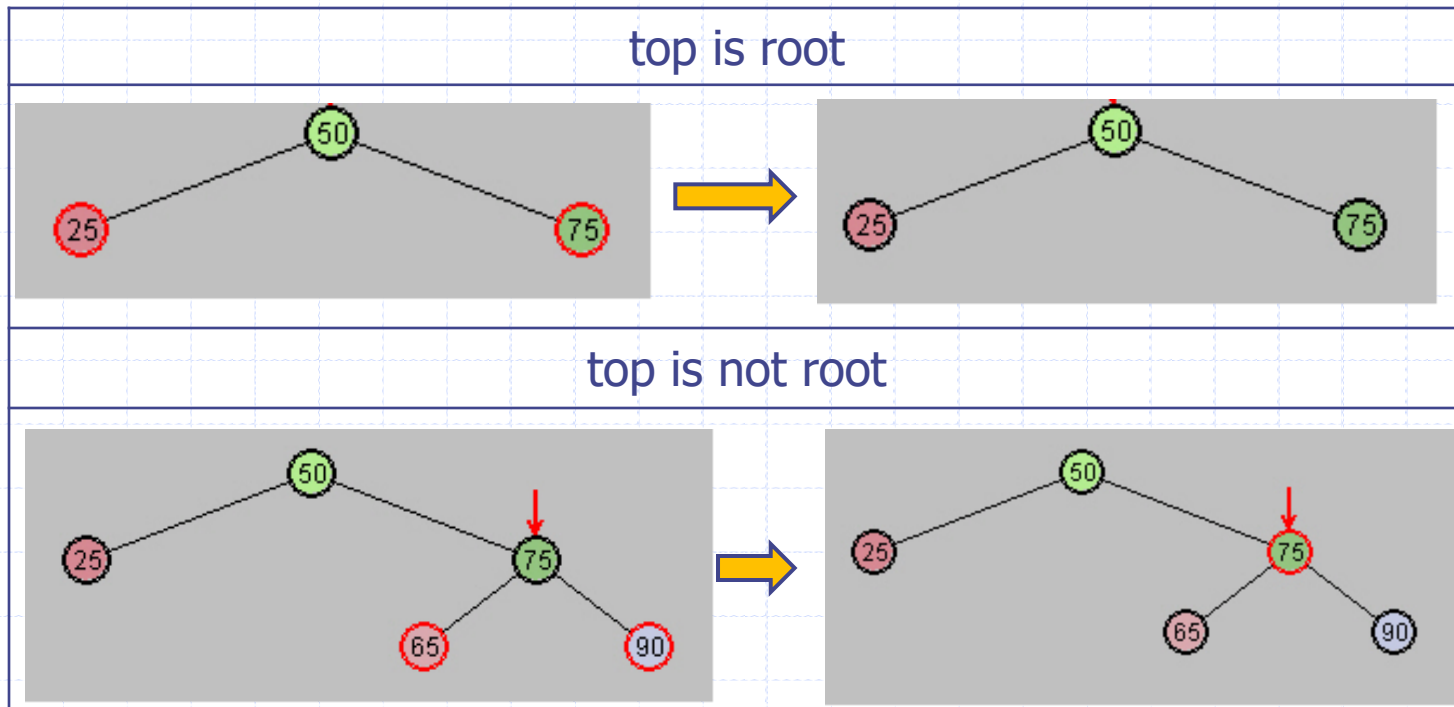
## 2. More detail:

- a. On the way down the tree perform color flips and rotations
- b. Insert new node
- c. Make further adjustments: color changes and rotations



# Color Flips On The Way Down

**Color Flip Strategy.** During the search for the insertion point, when a black node having two red children is encountered, a color flip is done. A color flip changes colors of all three nodes unless the top node is the root, in which case only the children's nodes change color.



# Color Flips On The Way Down

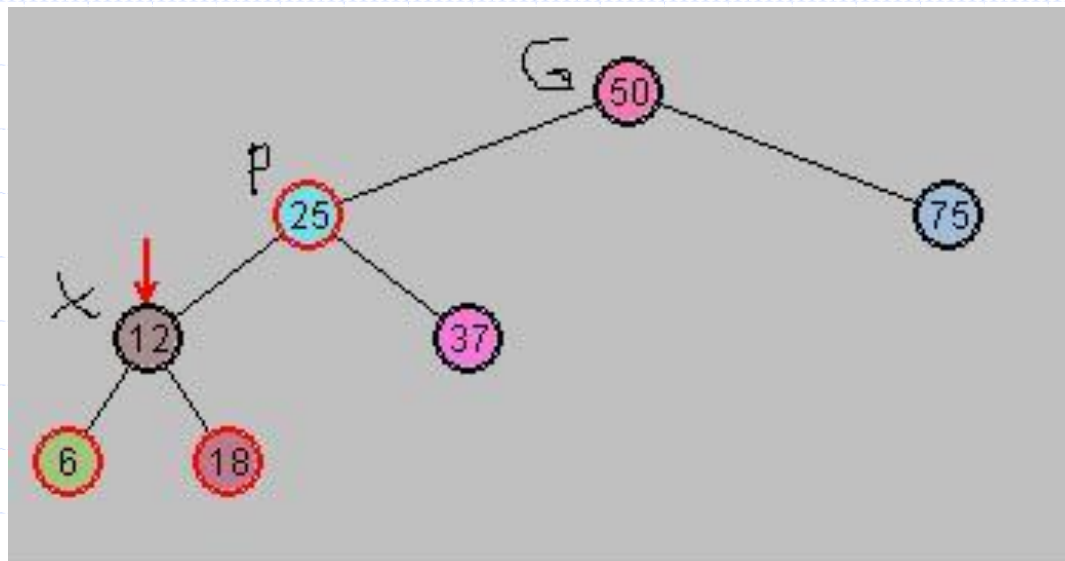
- ◆ This color-flip strategy is used as a preventive measure to make sure that adjustments needed after insertion don't propagate up the tree.
- ◆ After a color flip, all red-black tree properties continue to hold except for the property that red nodes must have black children.
- ◆ In particular, a color flip can introduce a *red-red violation*, which must be corrected to maintain red-black property. Before moving further to the insertion point, the violation must be corrected, using one or two *rotations*.

# Rotations On The Way Down

- ◆ These are done to correct red-red violations that occur from color flips. It can be shown that after a color flip, followed by one of the color change/rotation combinations mentioned below, the tree is once again red-black.
- ◆ Case 1 Outside grandchild causes a red-red violation after color flip. The Rule:
  - Change color of G
  - Change color of P
  - Rotate P, G in the direction that lifts X
- ◆ Case 2 The inside grandchild causes a red-red violation. The Rule:
  - Change color of G
  - Change color of X
  - Perform double rotation:
    - P, X, lifting X
    - G, X, lifting X

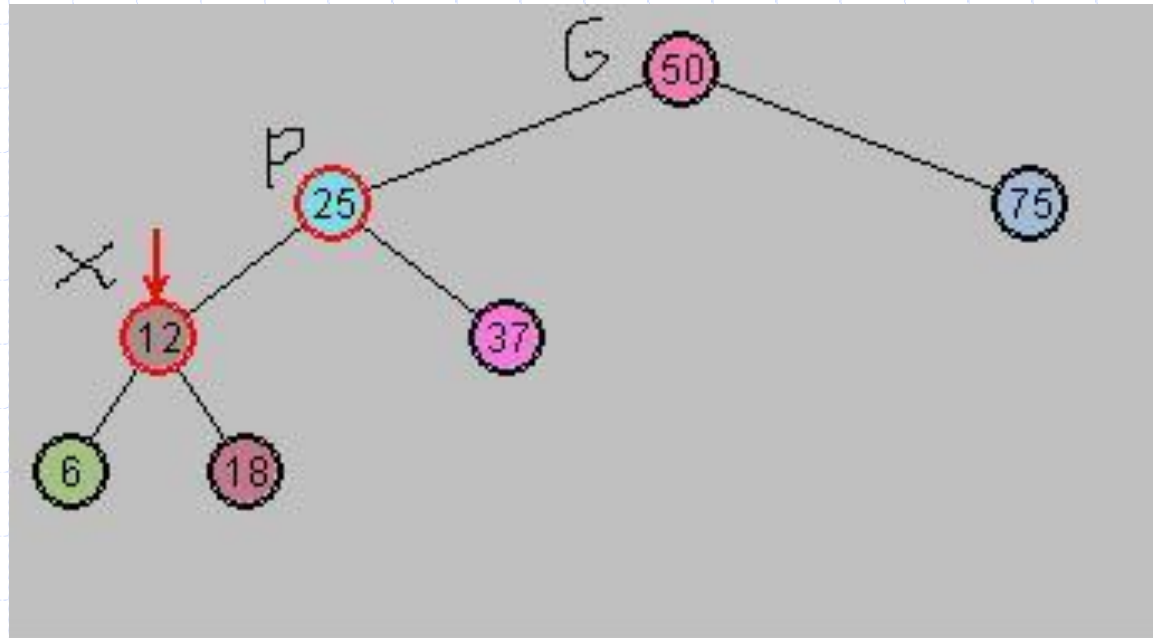
# Example for Case 1

- ◆ We wish to insert the node 3, but we first notice need for color flip



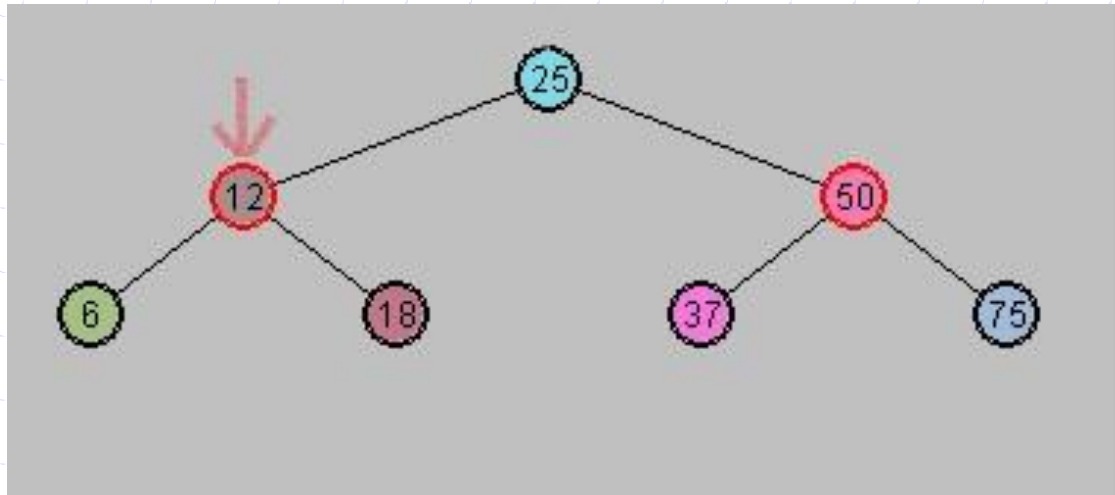
# Example for Case 1 - Continued

- ◆ After color flip, we encounter red-red violation caused by outer grandchild



# Example for Case 1 - Continued

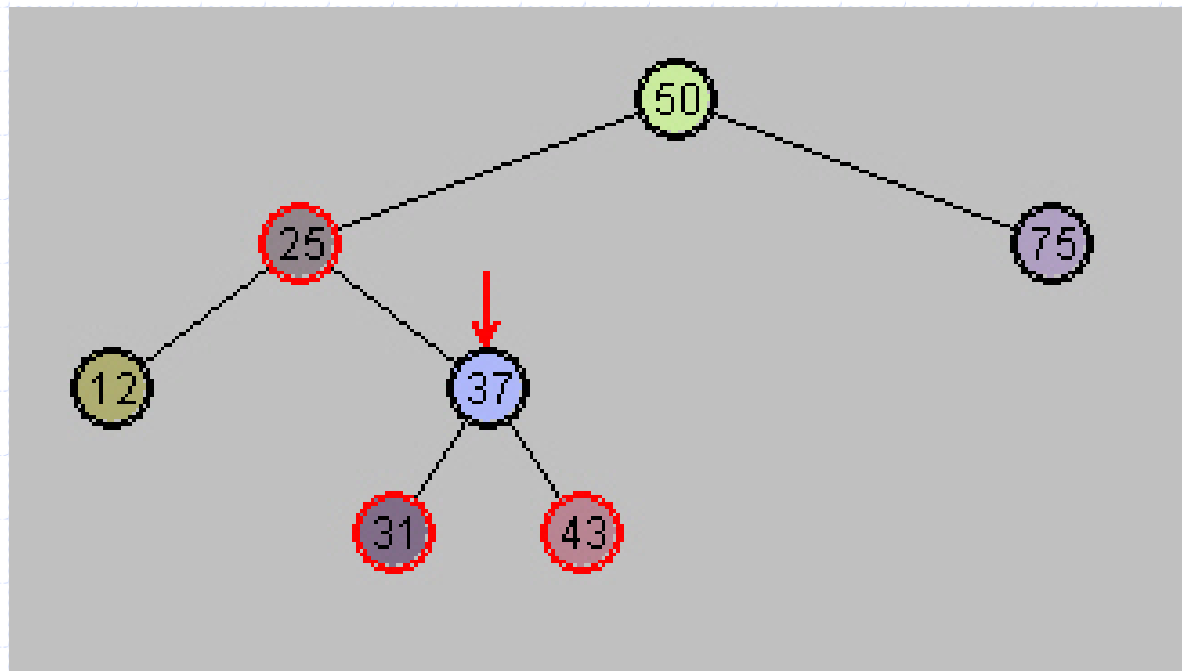
◆ We apply the rule for this case:



◆ Red-red violation has been eliminated and tree is now balanced.

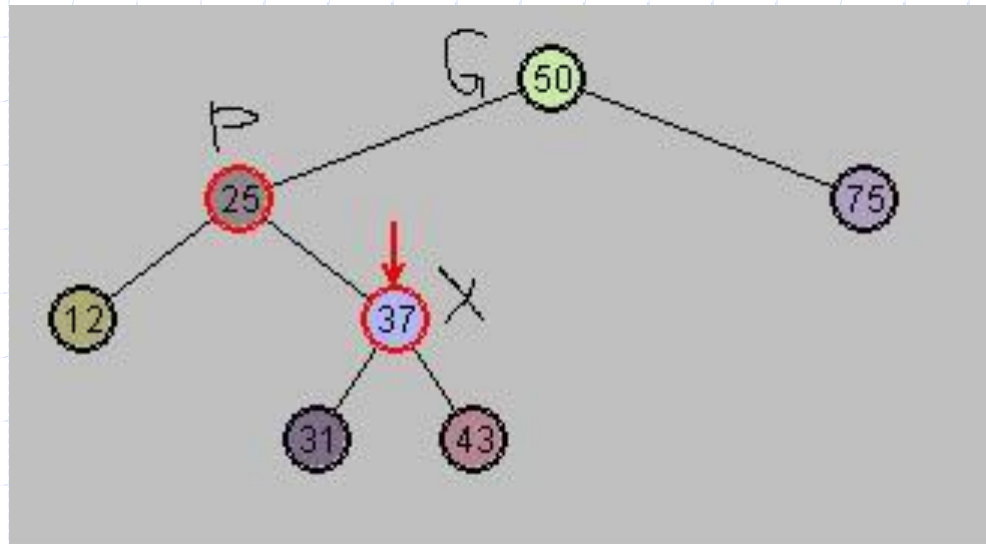
# Example for Case 2

- ◆ Suppose we are trying to insert the node 28.



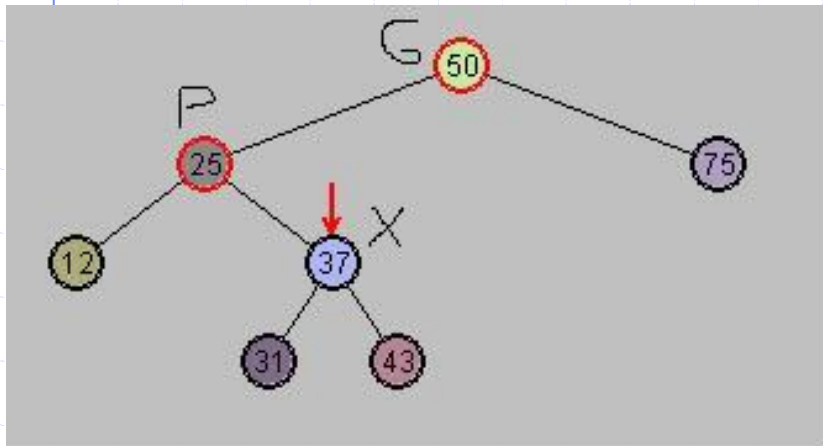
# Example for Case 2 - Continued

- ◆ Need to do a color flip which leads to a red-red violation caused by inner grandchild.



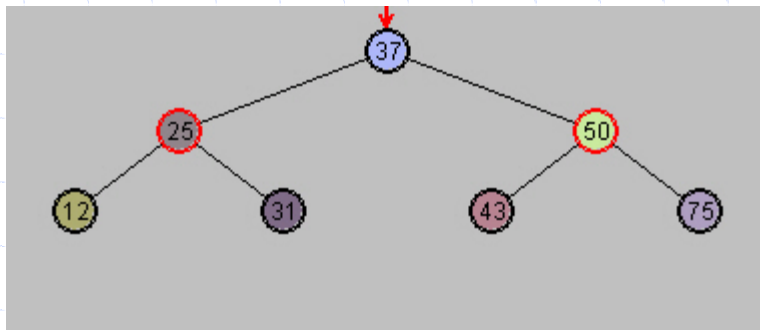
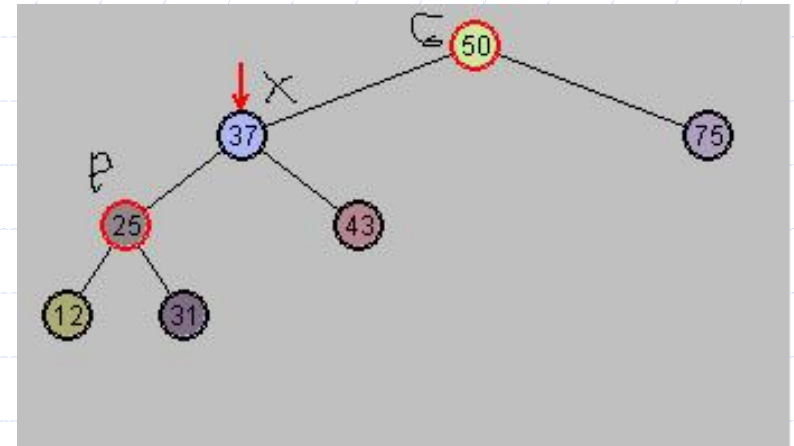


# Example for Case 2 - Continued



1. First change colors of G, X

2. Rotate X,P in the direction that lifts X.



3. rotate X, G in the direction that lifts X. Tree is now balanced. Can now insert 28.

# Insertion And Corrections

- ◆ Nodes are always inserted as red nodes. This can lead to further red-red violations.
- ◆ Three cases when new node X is inserted.
  - P is black, so no adjustment necessary (since X is red)
  - P is red and X is an outside grandchild. The Rule:

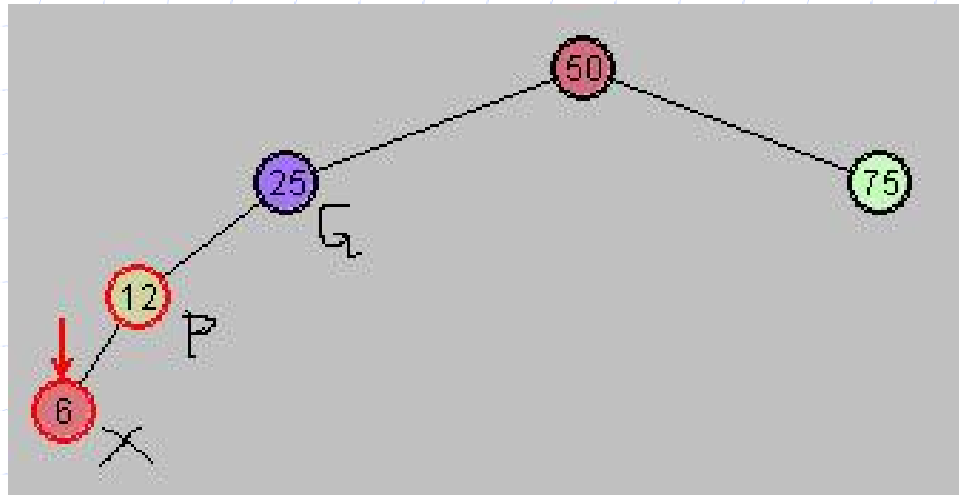
- Change color of G
- Change color of P
- Rotate P, G in direction that lifts X

- P is red and X is an inside grandchild. The Rule:

- Change color of G
- Change color of X
- Double rotation:
  - P, X, lifting X
  - G, X, lifting X

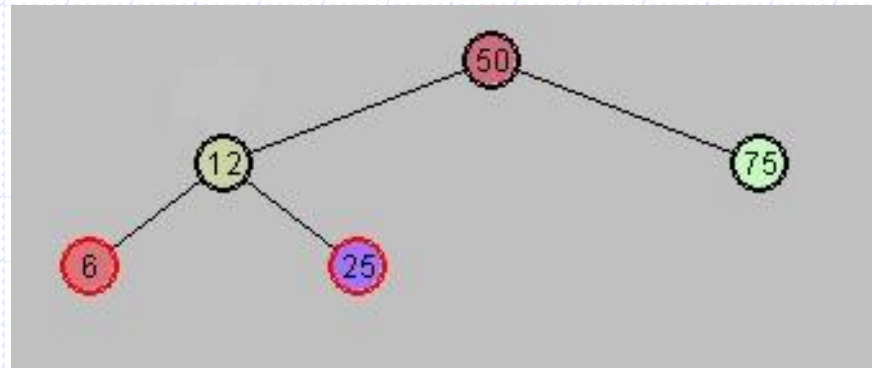
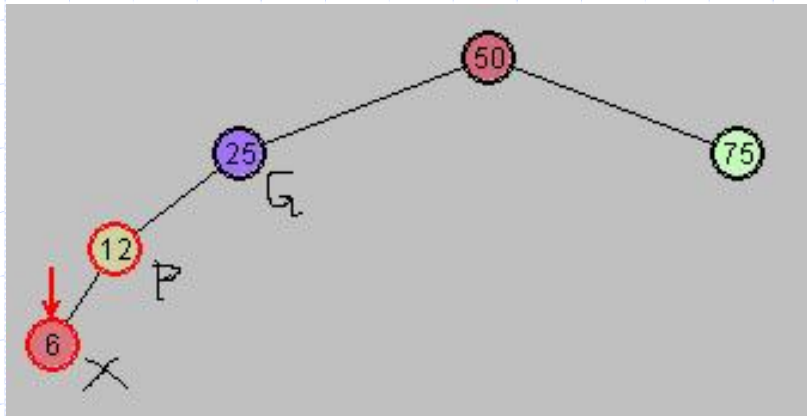
# Exercise - 1

- ◆ Insert 6 to the tree. Insertion results in outer grandchild red-red violation.



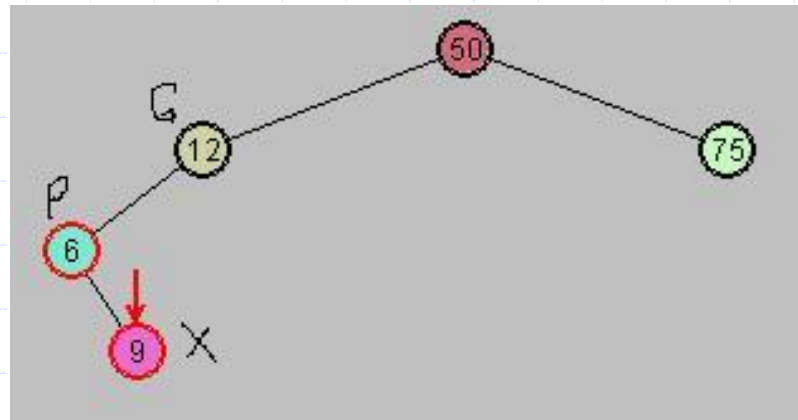
# Exercise - 1

- ◆ Insert 6 to the tree. Insertion results in outer grandchild red-red violation.



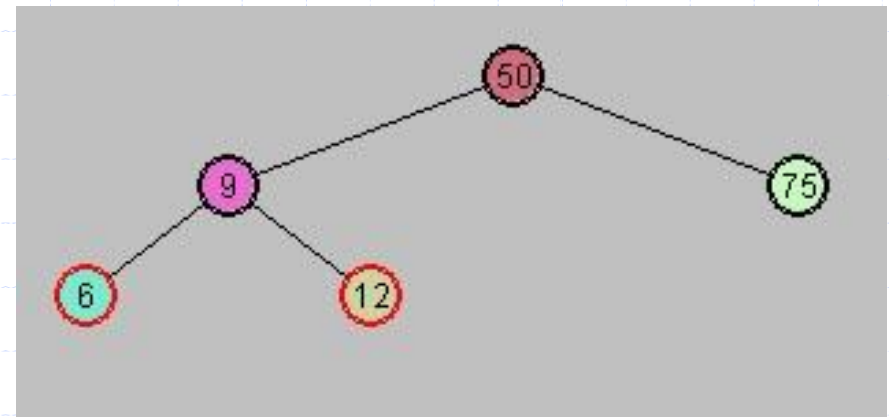
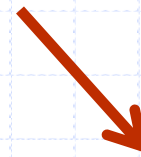
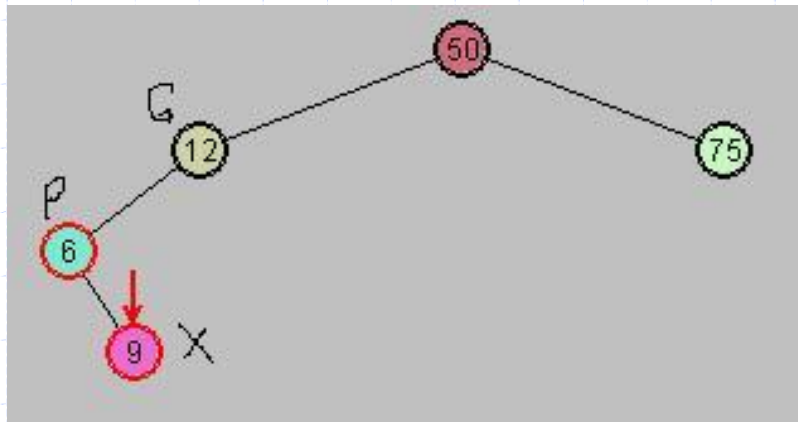
# Exercise - 2

- ◆ Insert 9 to the tree. Insertion results in inner grandchild red-red violation.



## Exercise - 2

- ◆ Insert 9 to the tree. Insertion results in inner grandchild red-red violation.



# Building Red-black Trees

- ◆ Use the insertion algorithm for red-black trees to successively insert the following nodes, starting with an empty tree.

1, 2, 3, 4, 5, 6

Show work on board.

# Conclusions

- ◆ Insertions of a node at depth  $d$  require
  - $O(d)$  to make flips and adjustments on the way to insertion point
  - $O(1)$  to make final color changes and adjustments
- ◆ Therefore, worst case running time for insertions is  $O(\log n)$
- ◆ Searches are done as in BSTs – as before, these require  $O(\log n)$
- ◆ Deletions can be shown also to take  $O(\log n)$ . Since the algorithm requires several cases and is a bit complicated, we do not go through it here.



# Main Point

The integrity of red-black trees is preserved after tree operations (insertions and deletions) are performed by maintaining the balance condition after execution of each operation. This maintenance does not increase the cost of operations because it requires only constant time, involving local color changes, color flips, and rotations.

Science of Consciousness: The ability to maintain its fundamental character in the face of change is the expression of the *invincible* quality of pure consciousness. Pure consciousness, in giving rise to diversity, maintains its unbounded and immortal status. In society, this invincible quality is seen when a small percentage of a population engages in group practice of the TM and TM-Sidhi Programs – the inherent harmony of the society is enlivened to the extent that it “averts the birth of an enemy.”

## Connecting the Parts of Knowledge With the Wholeness of Knowledge

### Balanced BSTs

1. A Binary Search Tree can be used to maintain data in sorted order more efficiently than is possible using any kind of list. Average case running time for insertions and searches is  $O(\log n)$ .
2. In a Binary Search Tree that does not incorporate procedures to maintain balance, insertions, deletions and searches all have a worst-case running time of  $\Omega(n)$ . By incorporating balance conditions, the worst case can be improved to  $O(\log n)$ .
3. *Transcendental Consciousness* is the field of perfect balance. All differences have Transcendental Consciousness as their common source.
4. Impulses Within the Transcendental Field. The sequential unfoldment that occurs within pure consciousness and that lies at the basis of creation proceeds in such a way that each new expression remains fully connected to its source. In this way, the balance between the competing emerging forces is maintained.
5. Wholeness Moving Within Itself. In Unity Consciousness, balance between inner and outer has reached such a state of completion that the two are recognized as alternative viewpoints of a single unified wholeness.